Université de Bordeaux
4TIN514U

Licence S&TS Mention Informatique, S5
2024/2025

Programmation fonctionnelle
TP noté du mardi 10 Décembre
Durée : 1h30.

Le barème est donné à titre **indicatif**. Le sujet comporte 5 pages.

a) Télécharger le fichier `student.ml` depuis Moodle. Le travail doit être effectué dans ce fichier `student.ml` lequel doit être déposé sur Moodle à l'issue du TP. Ce fichier contient un barème indicatif.

b) Renseignez vos nom, prénom et numéro de groupe.

c) Pensez à sauver régulièrement votre travail.

d) Il est recommandé d'utiliser les fonctions du module `List` chaque fois que c'est approprié.

e) Il y a des exemples pour chaque fonction demandée.

---

1 Préléminaires

1. Écrire une fonction `remove_duplicates l` **récursive terminale** de type `'a list -> 'a list` qui *supprime* les doublons de la liste `l` à partir de la gauche.

```
# remove_duplicates;;
- : 'a list -> 'a list = <fun>
# remove_duplicates [2; 1; 3; 1; 4; 5; 2];;
- : int list = [3; 1; 4; 5; 2]
```

2. Écrire une fonction `pairs_flatten pairs` **récursive terminale** de type `('a * 'a) list -> 'a list` qui prend une liste de couples et retourne la liste des élément qui apparaissent dans les couples dans l'ordre. Exemple :

```
# pairs_flatten;;
- : ('a * 'a) list -> 'a list = <fun>
# pairs_flatten [(1, 2); (2, 3); (3,  4); (2, 3)];;
- : int list = [1; 2; 2; 3; 3; 4; 2; 3]
```

3. Écrire un prédicat `exists predicate l` qui dit si au moins un des éléments de la liste `l` vérifie le prédicat `predicate`.

```
# exists;;
- : ('a -> bool) -> 'a list -> bool = <fun>
# exists (fun n -> n mod 3 = 0) [1; 3];;
- : bool = true
# exists (fun n -> n mod 2 = 0) [1; 3];;
- : bool = false
```

On va travailler sur des graphes simples orientés dont les noeuds sont désignés par des entiers naturels.

2 Arcs

Un arc d'un graphe est un couple d'entiers (o, e) où o est le noeud origine et e le noeud extrémité de l'arc. On définit le type suivant pour représenter un arc.

```
type arc = A of int * int
```

4. Écrire la fonction constructeur `make_arc origin extremity` qui retourne l'arc $(origin, extremity)$ et les fonctions accesseur correspondantes `arc_origin arc` et `arc_extremity arc`. Exemples :

```
# make_arc;;
- : int -> int -> arc = <fun>
# make_arc 4 3;;
- : arc = A (4, 3)
# arc_origin;;
- : arc -> int = <fun>
# arc_extremity;;
- : arc -> int = <fun>
# arc_origin (make_arc 4 3);;
- : int = 4
# arc_extremity (make_arc 3 4);;
- : int = 3
```

5. Écrire une fonction `reverse_arc arc` qui retourne l'arc `arc` inversé. Exemple :

```
# reverse_arc;;
- : arc -> arc = <fun>
# reverse_arc (make_arc 3 5);;
- : arc = A (5, 3)
```

6. Écrire une fonction `pair_to_arc pair` qui construit un arc à partir d'une paire.

7. Écrire une fonction `arc_to_pair arc` qui retourne la paire contenue dans l'arc `arc`.

8. Écrire une fonction `pairs_to_arcs pairs` qui construit une liste d'arcs à partir d'une liste de paires.

9. Écrire une fonction `arcs_to_pairs arcs` qui construit la liste des paires conenues dans les arcs `arcs`. Exemples :

```
# pair_to_arc;;
- : int * int -> arc = <fun>
# pair_to_arc (1, 2);;
- : arc = A (1, 2)
# arc_to_pair;;
- : arc -> int * int = <fun>
# arc_to_pair(pair_to_arc (1, 2));;
- : int * int = (1, 2)
# pairs_to_arcs;;
- : (int * int) list -> arc list = <fun>
# let pairs = [(1, 2); (1, 3); (3, 4); (4, 1); (5, 6)];;
```

```
# let arcs = pairs_to_arcs pairs;;
val arcs : arc list = [A (1, 2); A (1, 3); A (3, 4); A (4, 1); A (5, 6)]
# arcs_to_pairs;;
- : arc list -> (int * int) list = <fun>
# arcs_to_pairs arcs;;
- : (int * int) list = [(1, 2); (1, 3); (3, 4); (4, 1); (5, 6)]
```

3 Graphes

On utilise le type `graph` suivant pour représenter les graphes simples orientés. Le champ `isolated_nodes` contient les noeuds isolés (qui ne sont connectés à aucun arc) et le champ `arcs` la liste des arcs du graphe.

```
type graph = { arcs : arc list; isolated_nodes : int list }
```

10. Écrire la fonction constructeur `make_graph arcs isolated_nodes` qui crée un graphe à partir d'une liste d'arcs et une liste de noeuds isolés ainsi que les fonctions accesseur correspondantes `graph_arcs graph` et `graph_isolated_nodes graph`. **On supposera les arguments de `make_graph` corrects ; c'est-à-dire que les noeuds isolés n'apparaissent pas dans les arcs.** Exemples :

```
# let mygraph = make_graph arcs [7];;
val mygraph : graph =
  {arcs = [A (1, 2); A (1, 3); A (3, 4); A (4, 1); A (5, 6)];
   isolated_nodes = [7]}
# graph_arcs mygraph;;
- : arc list = [A (1, 2); A (1, 3); A (3, 4); A (4, 1); A (5, 6)]
# graph_isolated_nodes mygraph;;
- : int list = [7]
```

11. Créer le graphe `mygraph` défini dans l'exemple prédédent.

12. **Facultatif.** Charger le fichier `dot.ml` et visualiser le graphe `mygraph` avec `graph_view`.

```
# graph_view mygraph;;
```

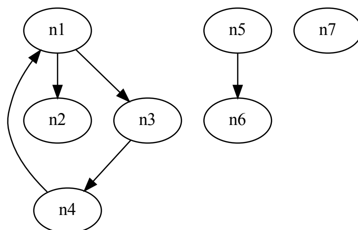

Fig. 1 : Représentation de `mygraph`

13. Écrire une fonction `nodes_from_arcs arcs` qui retourne la liste des noeuds présents dans les arcs de la liste d'arcs `arcs` **sans doublons**. L'ordre n'a pas d'importance. Exemples :

```
# nodes_from_arcs;;
- : arc list -> int list = <fun>
# nodes_from_arcs (graph_arcs mygraph)
- : int list = [2; 3; 4; 1; 5; 6]
```

14. Écrire une fonction `graph_nodes graph` qui retourne la liste de tous les noeuds du graphe. Exemple :

```
# graph_nodes mygraph;;
- : int list = [2; 3; 4; 1; 5; 6; 7]
```

15. Écrire une fonction `node_neighbours node arcs` qui donne les voisins du noeud `node` par les arcs de la liste `arcs`. Exemples :

```
# node_neighbours 1 (graph_arcs mygraph);;
- : int list = [2; 3]
# node_neighbours 5 (graph_arcs mygraph);;
- : int list = [6]
```

On souhaite simuler le marquage des noeuds accessibles à partir d'un certain noeud d'un graphe.

16. Écrire une fonction `mark_from_nodes nodes arcs marked` qui retourne la liste des noeuds déjà marqués `marked`, complétée avec les noeuds accessibles à partir des noeuds de la liste `nodes`.

17. En utilisant la fonction `mark_from_nodes`, écrire une fonction `mark_from_node node arcs` qui retourne la liste des noeuds accessibles à partir du noeud `node` en suivant des arcs de la liste `arcs`. Le noeud initial `node` **ne sera pas dans la liste retournée sauf s'il est accessible en traversant au moins un arc**.

```
# mark_from_nodes;;
- : int list -> arc list -> int list -> int list = <fun>
# mark_from_nodes [1; 5] (graph_arcs mygraph) [];;
- : int list = [1; 2; 3; 4; 5; 6]
# mark_from_nodes [1; 5] (graph_arcs mygraph) [8; 9];;
- : int list = [9; 8; 1; 2; 3; 4; 5; 6]
# mark_from_node;;
- : int -> arc list -> int list = <fun>
# mark_from_node 1 (graph_arcs mygraph);;
- : int list = [2; 3; 4; 1] (* 1 est dans la liste car accessible à partir de 4 *)
# mark_from_node 5 (graph_arcs mygraph);;
- : int list = [6] (* 5 n'est pas dans la liste *)
```

18. Écrire une fonction `unorient_arcs arcs` qui retourne la liste des arcs `arcs` complétée avec les arcs inverses. Exemples :

```
# unorient_arcs;;
- : arc list -> arc list = <fun>
# unorient_arcs (graph_arcs mygraph);;
- : arc list =
[A (5, 6); A (6, 5); A (4, 1); A (1, 4); A (3, 4); A (4, 3); A (1, 3);
 A (3, 1); A (1, 2); A (2, 1)]
```

19. Écrire un prédicat `graph_connected_p graph` qui dit si la version non orientée du graphe `graph` est connexe c'est-à-dire si tous les noeuds sont connectés entre eux après effacement de l'orientation. On pourra vérifier que tous les noeuds sont accessibles à partir d'un noeud quelconque à partir des arcs du graphes non orientés (ou autrement dit à partir des arcs orientés dans les deux sens).

```
# graph_connected_p g3;;
- : bool = true
# graph_connected_p mygraph;;
- : bool = false
```

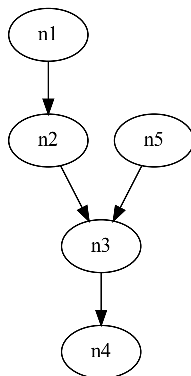

FIG. 2 : Représentation de `g3`

20. Écrire un prédicat `graph_has_circuit_p graph` qui dit si un graphe `graph` contient un circuit non vide (contenant au moins un arc) d'un noeud vers lui-même.

```
# graph_has_circuit_p g3;;
- : bool = false
# graph_has_circuit_p mygraph;;
- : bool = true
# graph_has_circuit_p (make_graph [] []);;
- : bool = false
```